Your search matched **1** of **990765** documents.

A maximum of **1** results are displayed, **15** to a page, sorted by **Relevance** in **descending** order.
You may refine your search by editing the current search expression or entering a new one the text box.
Then click **Search Again**.

(heap) <paragraph> (linked list) Search Again

Results:
Journal or Magazine = **JNL** Conference = **CNF** Standard = **STD**

1 **An efficient implementation of Ada delay**
Gearhart, L.M.;
Aerospace and Electronics Conference, 1990. NAECON 1990., Proceedings of th 1990 National , 21-25 May 1990
Page(s): 603 -605 vol.2

[Abstract] [PDF Full-Text (204 KB)] **IEEE CNF**

AN EFFICIENT IMPLEMENTATION OF ADA DELAY

Larry M. Gearhart

TRW
Military Electronics and Avionics Division
Dayton Engineering Laboratory
4021 Executive Drive
Dayton, OH 45430

Abstract

This paper develops an approach to efficient implementation of Ada delay. It is a preemption-oriented implementation which guarantees a response to a high priority task within a period of time proportional to the logarithm of the number of delayed tasks. Two versions are proposed. Both versions avoid the usual execution overhead of balanced tree algorithms (such as AVL trees) or the linear time-complexity growth associated with linked lists. A static version is proposed which is optimized for execution efficiency and a dynamic version is optimized for flexibility. Both versions are an adaptation of the standard heap sort algorithm, with extensions designed to allow efficient delay cancellation.

Introduction

Embedded avionics software must often respond to external events within stringent timing constraints. Some tasks must execute periodically. Others may be required to grant only a brief window of time to waiting for the arrival of data that may never come. Finally, a task may be required to execute at a specific clock time.

The Ada DELAY statement is not the sole means of controlling the timing of events in an Ada application. Nevertheless, it is the only means defined by the language. Thus, Ada applications that must be portable will rely on it exclusively. Even most applications for which execution time is more critical than portability should rely on this statement provided the compiler implementation is sensitive to real-time issues.

It is therefore important for compilers to provide an efficient implementation of Ada DELAY. Unfortunately, the Ada standard levys a weak requirement on compiler support for DELAY semantics. The standard does not require that expiration of a DELAY immediately results in making a task eligible for execution. The execution of a delay statement suspends further execution of the task that executes the delay statement, for at least the duration specified.

See paragraph 9.6.1 in the LRM ([1]). One would prefer that the delay expiration would cause the task to become eligible for execution soon after expiration of the delay. However, the LRM does not spell this out. The Approved Ada Language Commentaries say, in further elaboration:

The accuracy of the delay imposed by a delay statement is not directly related to the value of SYSTEM.TICK nor to the value of DURATION'SMALL since the value of SYSTEM.TICK only reflects the frequency with which successive calls to CLOCK can be expected to change, and the value of DURATION'SMALL only reflects the accuracy with which values of type DURATION can be represented. Of course, since the clock used for the function CLOCK also can be used to schedule delay statements, it can be expected that in a reasonable implementation, delay statements will be executed with an accuracy that is no worse than SYSTEM.TICK. An accuracy that is significantly worse would require justification in terms of AI-00325.

See AI-00201/07 in [2]. Because of this caveat, the delay may be scheduled to expire at a time slightly prior to the time the programmer might expect, due to inaccuracy in the timer. This requirement is deliberately loose to allow implementation in a variety of architectures and for a variety of application arenas. One may term the accuracy problem "duration uncertainty," while the problem of lax response to delay expiration may be termed the latency problem of Ada DELAY, although the problems are as much an issue of hardware as software support.

Even when the compiler vendor is sensitive to the DELAY latency issue, an efficient implementation is not always provided. The most obvious implementation (such as a linear list) is usually chosen in order to get the product out the door. Application programmers have been justifiably disheartened by this state of affairs. Yet they themselves have not provided the impetus needed to encourage vendors to pay attention to the issue. They have usually been satisfied to bypass the construct altogether and implement their own "timer services."

This paper describes an approach to an efficient implementation of Ada delay that is not subject to the usual difficulties of linear cost growth or excessive overhead. The implementation is based upon the heap data structure, with added functions to allow for DELAY cancellation.

Requirements

In the best case, an implementation of Ada DELAY will have, at most, logarithmic growth in execution cost, as the number of pending delays increases, but without imposing excessive overhead when the number of pending delays is small. The implementation must make use of a timer interrupt to ensure that latency, in the sense described above, is not an issue. When a task requests a positive DELAY, it should be suspended and the delay value should be compared with that of the timer. If the new delay is smaller, it should replace the timer value. In any case, the new delay request should be collected with the other pending requests, preferably in order according to delay-expiration clock time.

When the timer delay expires, the earliest pending delay has expired. That delay request is then removed from the collection (with the requesting task becoming eligible for execution) and the next one in line determines the new timer delay. Thus removal of the top request should result naturally in the next request being the nevest "earliest delay."

CH2881-1/90/0000-0603 $1.00 © 1990 IEEE

Ada allovs cancellation of a DELAY in several contexts, including selective vait and abort. If a DELAY should happen to be canceled, the associated request should be removed from the collection without disturbing the order of the remaining requests. The next request in line replaces the canceled one in the timer, if necessary.

Implementation

Absolute (clock-time), rather than relative (counter), time determines the most natural ordering on the requests. Thus, the earliest request is the one whose delay is scheduled to expire at a clock time prior to that of the other requests. The natural time reference for the delay timer, on the other hand, is relative time. That is, the delay timer can simply function as a count-down circuit with "counts" scaled to DURATION.

In programs with fev tasks, execution cost growth is not an issue, although latency still is. Any implementation, such as maintaining a sorted linear list, will be adequately efficient. When the number of pending delays becomes large, searching through a linear list becomes prohibitively expensive. One could conceive of using a balanced tree to keep track of earliest expirations, but the known implementations require excessive overhead. A third possibility, described in this paper, is to use a heap. If delays could not be canceled for any reason, this would clearly be the best approach. See [3] for a clear discussion of heap data structure primitives.

There are three actions associated with the "collection" of requests:

1. Insert pending request
2. Remove earliest request
3. Remove canceled request

The efficiency requirement on DELAY translates to a requirement on the execution cost of these three actions. The execution time of the actions should not increase worse than logarithmically with the number of pending delays, and the execution time for very fev pending delays should be comparable to that of a linear implementation.

All that is necessary, therefore, is to augment the heap approach with an action to remove an entry without disturbing the heap order on the remaining entries. Also, when a nev request is inserted into the heap, its place in the heap should be "remembered" so that if the request must be canceled, the entry may be accessed and removed.

Standard Heap Version

Consider the standard heap implementation in terms of an array. Assume for the moment that, as will be shown, one can remove an entry buried in the heap without violating the heap order. Suppose that the array index of an entry, D1, is used as a pointer to access the entry, and that this pointer is saved when the entry is first inserted into the heap. Then when a nev entry, D2, is inserted or removed, D1 may be moved to a nev location, thus invalidating the "pointer" to D1.

One way around this problem is to maintain a second array of "pointers". Each heap entry "owner" saves an index into this array. The array entry pointed to by this index points to the owner's heap entry. Each heap entry maintains an index to its own "pointer" and updates that pointer when it gets moved. This, naturally, adds overhead to the heap insert and remove-first operations. See Figure 1.

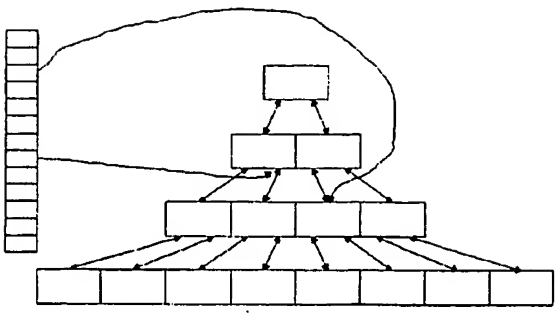

Figure 1. Conventional Heap with Safe Pointers

Naturally, this implementation limits the total number of pending delays possible by the sizes of the arrays. The next implementation to be considered goes one step further and uses dynamic memory.

Dynamic Heap Version

Consider Figure 2. In this implementation, each heap entry is a dynamically allocated cell. Each cell maintains enough pointers, as shown in the figure, to knov its position in the heap relative to its neighbors. In this version, the overhead of the standard insert and remove-first operations is somevhat greater, but still compares very favorably with, say, a balanced tree implementation. Of course the auxiliary array is no longer needed, since the cells don't change contents, the cells simply change position in the heap during "insert" and "remove-first".

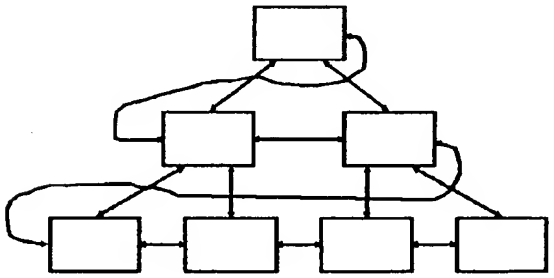

Figure 2. Dynamic Heap Data Structure

While the standard heap makes an array look like a tree, this version makes a linked list look like a tree. In both versions, the entry at the end of the array or list plays a special role in insert and remove-first operations. That entry also plays a special role in "cancel".

Nov consider the cancel operation. The basic idea is to remove the selected entry and then rearrange O(log(n)) remaining entries so that the heap property is preserved. The heap property is "Every entry is greater than or equal to its sons." When an entry is removed it leaves a hole in the heap which must be filled. One obvious candidate to plug up the hole is the last entry in the array or list. If the nev entry is less than one of its nev sons, then it changes places with the greater of those sons, and continues to migrate downward.

If the nev entry is greater than both sons, it may also be greater than its nev parent. In this case it must migrate upvard. In both cases, the nev entry exchanges places with a parent or child in order to reestablish its place in the heap. This process is so like the insert and remove-first operations it should need no further elaboration.

Conclusions

Three essential operations have been identified for the efficient implementation of Ada DELAY. The heap data structure has been identified as the natural underlying structure for these operations. Two versions of the heap have been put forvard, and the implementation of "cancel" has been described in both.

One hopes that this discussion will generate a new level of interest, in the Ada compiler user and vendor communities, in the serious application of Ada DELAY. While the suggestions presented here do not solve all of the problems cited in the literature (e.g., an absolute delay might be more useful than relative delay in many applications), it is hoped that the proposed implementation will prove fruitful in general applications.

References

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983.

2. Approved Ada Language Commentaries, Ada Letters, Volume IX, Number 3, Spring 1989.

3. Robert Sedgewick, Algorithms, Addison-Wesley Publishing Company, Inc., Reading MA., 1983.